5.4 Aliasing and "Mutation at a Distance"

Through our new object-based memory model, we've seen that the Python interpreter associates each variable with the *id* of an object. There is nothing stopping two or more variables from containing the same id, which means that two variables can refer to the same object. This causes some interesting situations when more than one variable refers to the same mutable object. In this section, we will use our memory model to better understand this specific (and common) situation.

Aliasing

Let `v1` and `v2` be Python variables. We saw that `v1` and `v2` are **aliases** when they refer to the same object.[1]

[1] The word "alias" is commonly used when a person is also known under a different name. For example, we might say "Eric Blair, alias George Orwell." We have two names for the same thing, in this case a person.

Consider the following Python code:

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x
```

`x` and `z` are aliases, as they both reference the same object. As a result, they have the same id. You should think of the assignment statement `z = x` as saying "make `z` refer to the object that `x` refers to." After doing so, they have the same id.

```
>>> id(x)
4401298824
>>> id(z)
4401298824
```

In contrast, `x` and `y` are not aliases. They each refer to a `list` object with `[1, 2, 3]` as its value, but they are two different list objects, stored separately in your computer's memory. This is again reflected in their different ids.

```
>>> id(x)
4401298824
```

```
>>> id(y)
4404546056
```

Here is the state of memory after the code executes:

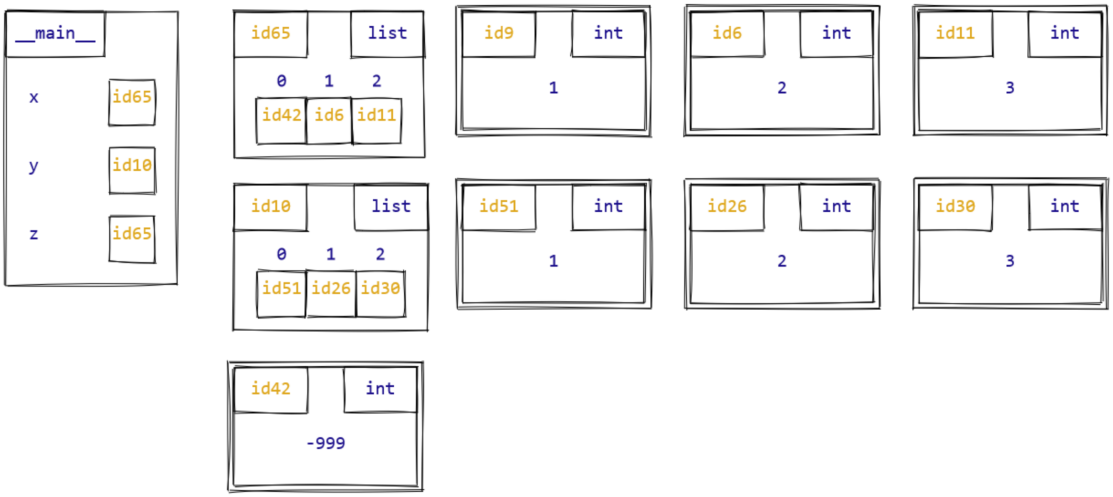
Aliasing and mutation

Aliasing is often a source of confusion for programmers because it allows "mutation at a distance": the modification of a variable's value without explicitly mentioning that variable. Here's an example:

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x
>>> z[0] = -999
>>> x   # What is the value?
```

The statement `x[0] = -999` line mutates the value of `z`. But without ever mentioning `x`, it also mutates the value of `x`!

Imprecise language can lead us into misunderstanding the code. We said above that "the third line mutates the value of `z`". To be more precise, the third line mutates the object that `z` refers to. Of course we can also say that it mutates the object that `x` refers to—they are the same object.

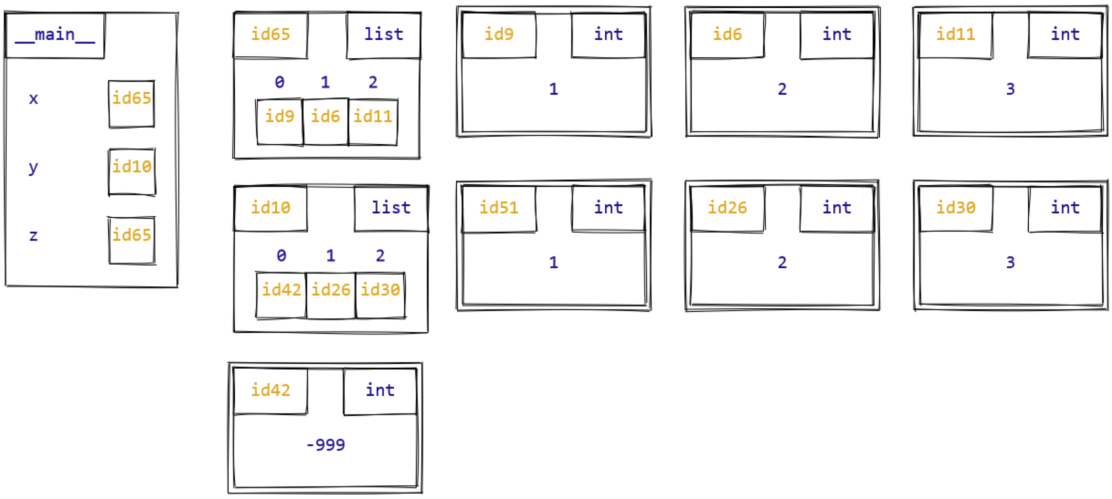


The key thing to notice about this example is that just by looking at the line of code, `z[0] = -999`, you can't tell that `x` has changed. You need to know that on a previous line, `z` was made an alias of `x`. This is why you have to be careful when aliasing occurs.

Contrast the previous code with this sequence of statements instead:

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x
>>> y[0] = -999
>>> x   # What is the value?
```

Can you predict the value of `x` on the last line? Here, the third line mutates the object that `y` refers to, but because it is not the same object that `x` refers to, we still see `[1, 2, 3]` if we evaluate `x`. Here's the state of memory after these lines execute:



Variable reassignment, again

What if we did this instead?

```
>>> x = (1, 2, 3)
>>> z = x
>>> z = (1, 2, 3, 40)
>>> x   # What is the value?
```

Again, we have made x and z refer to the same object. So when we change z on the third line, does x also change? This time, the answer is an emphatic **no**, and it is because of the kind of change we make on the third line. Instead of mutating the object that z refers to, we reassign z refer to a new object. This obviously can have no effect on the object that x refers to (or *any* object). Even if we switched the example from using immutable tuples to using mutable lists, x would be unchanged.

Given two aliases x and z, if we reassign x to a new object, that has *no effect* on z. We say that reassigning x *breaks the aliasing*, as afterwards x and z no longer refer to the same object, and so are no longer aliases.

Aliasing and loop variables

In Chapter 4, we saw two types of loops: element-based and index-based for loops. With index-based loops, the loop variable referred to an

integer object that could be used as an index to a collection (typically a `list`). But in element-based for loops, the loop variable is an alias to one of the objects *within* the collection. Suppose we have the following element-based for loop:

```
>>> numbers = [5, 6, 7]
>>> for number in numbers:
...     number = number + 1
...
>>> numbers
[5, 6, 7]
```

Notice how the values in the list `numbers` did not change (i.e., the for loop did not mutate `numbers`). This is because the loop variable `number` is an alias for the integer objects found inside `numbers`. The assignment statement inside the for loop simply changes what the loop variable refers to, but does not change what the contents of the list `numbers` refers to. If we would like to increment each object contained in the list, we must use an index-based for loop:

```
>>> numbers
[5, 6, 7]
>>> for i in range(0, len(numbers)):
...     numbers[i] = numbers[i] + 1
...
>>> numbers
[6, 7, 8]
```

The assignment statement in the index-based for loop is fundamentally different from the assignment statement in the element-based for loop. Statements of the form `<name> = _____` are *reassign* the variable `<name>` to a new value. But assignment statements of the form `<name>[<index>] = _____` *mutate* the list object that `<name>` currently refers to.

Two types of equality

Let's look one more time at this code:

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x
>>> id(x)
4401298824
>>> id(y)
4404546056
>>> id(z)
4401298824
```

What if we wanted to see whether `x` and `y`, for instance, were the same? Well, we'd need to define precisely what we mean by "the same". Our familiar `==` operator checks whether two objects have the same *value*. This is called **value equality**.

```
>>> x == y
True
>>> x == z
True
```

But there is another Python operator, `is`, which checks whether two objects have the same *ids*. This is called **identity equality**.

```
>>> x is y
False
>>> x is z
True
```

Identity equality is a *stronger* property than value equality: for all objects `a` and `b`, if `a is b` then `a == b`.[2] The converse is not true, as we see in the above example: `a == b` does not imply `a is b`.

[2] In Python it is technically possible to change the behaviour of `==` in unexpected ways (like always returning `False`), but this is a poor programming practice and we won't consider it in this course.

Aliasing with immutable data types

Aliasing also exists for immutable data types, but in this case there is never any "action at a distance", precisely because immutable values can never change. In the example below, `x` and `z` are aliases of a tuple object. It is impossible to modify `x`'s value by mutating the object `z` refers to, since we can't mutate tuples at all.

```
>>> x = (1, 2, 3)
>>> z = x
>>> z[0] = -999
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Automatic aliasing of (some) immutable objects

The above discussion actually has a very interesting implication for how we reason about variables referring to immutable objects: *if two variables have the same immutable value, the program's behaviour does not depend on whether the two variables are aliases or not*.

For example, consider the following two code snippets:

```
>>> x = (1, 2, 3)
>>> y = (1, 2, 3)
>>> my_function(x, y)
10
```

```
>>> x = (1, 2, 3)
>>> y = x
>>> my_function(x, y)
10
```

These two code snippets will always behave the same way, regardless of what `my_function` actually does! Because `x` and `y` refer to immutable values, the behaviour of `my_function` depends only on the values of the object, and not their ids.

This allows the Python interpreter to save a bit of computer memory by not creating new objects for some immutable values. For example, every

occurrence of the boolean value `True` refers to the same object:

```
>>> id(True)
1734328640
>>> x = True
>>> id(True)
1734328640
>>> id(10 > 3)
1734328640
>>> id(not False)
1734328640
```

A bit more surprisingly, "small" integers are automatically aliased, while "large" integers are not:

```
>>> x = 43
>>> y = 43
>>> x is y
True
>>> id(x)
1734453840
>>> id(y)
1734453840
>>> a = 1000
>>> b = 1000
>>> a is b
False
>>> id(a)
16727840
>>> id(b)
16727856
```

The other immutable data type where the Python interpret takes this object creation "shortcut" is with *some* string values:

```
>>> name1 = 'David'
>>> name2 = 'David'
>>> name1 is name2
True
>>> full_name1 = 'David Liu'
>>> full_name2 = 'David Liu'
>>> full_name1 is full_name2
False
```

The exact rules for when the Python interpreter does and does not take this shortcut are beyond the scope of this course, and actually change from one version of Python to the next. For the purpose of writing Python code and doing object comparisons, the bottom line is:

- For *boolean* values, use `is` to compare for equality.[3]

  [3] Though also keep in mind that you should never write `<expr> is True` or `<expr> is False`, since these are equivalent to the simpler `<expr>` and `not <expr>`, respectively.

- For *non-boolean immutable* values, use `==` to compare for equality, as using `is` can lead to surprsing results.
- For *mutable* values, use `==` to compare value equality (almost always what you want).
- For *mutable* values, use `is` to check for aliasing (almost never what you want).